Data structures for buffering

Roland Sipos for DUNE DAQ

DUNE DAQ/SC Meeting
7th June 2021

Overview

- Buffering
    - Requirements and constraints
    - Latency Buffer
- Data structures
    - Classification
    - Few examples
- Use-cases in the “readout” package
    - WIB LB: lock-free ProducerConsumerQueue from Folly
    - PDS LB: ConcurrentSkipList from Folly
- Other potential use-cases
- Outlook

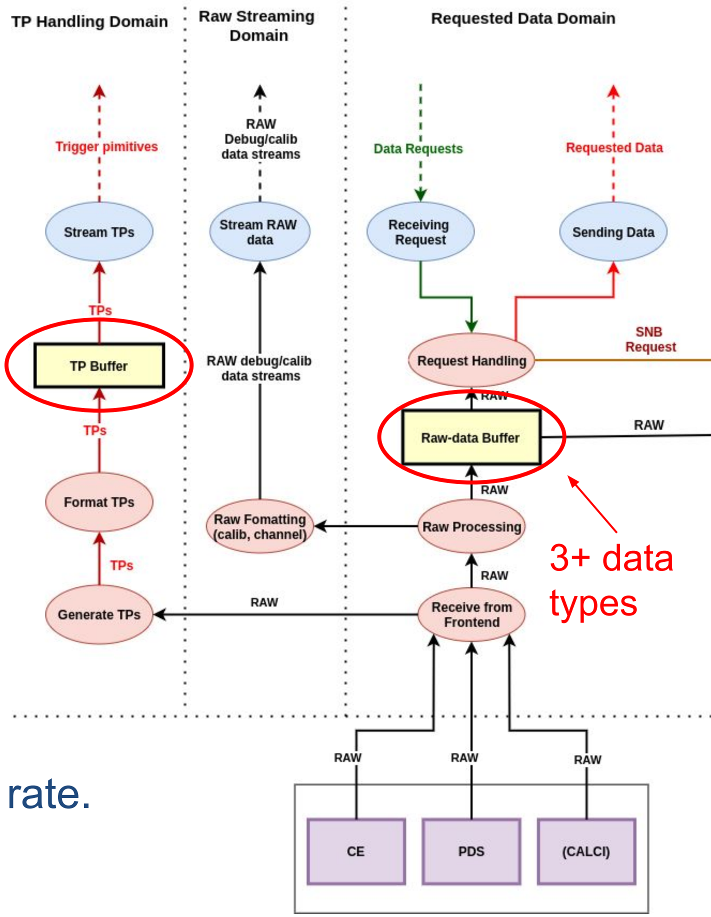
Buffering

Buffering

- A **data buffer** is a region of memory storage used to temporarily store data while it is being moved from a source to a destination.

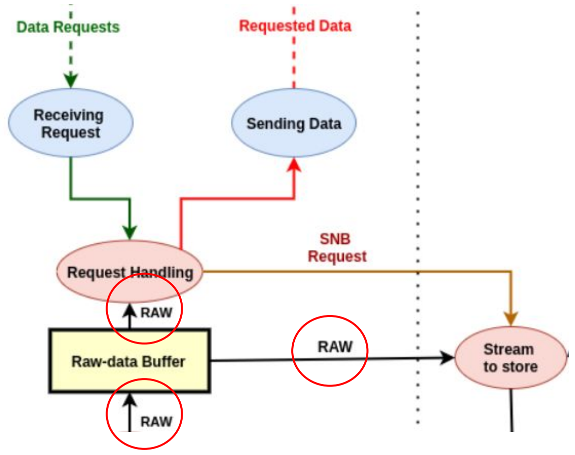
- Buffers are **typically used when** there is a difference between the rate at which data is received and the rate at which it can be processed, or when these rates are variable.

- A buffer often adjust “timing” by implementing a queue algorithm in memory, simultaneously writing data into the queue at one rate and reading it at another rate.
  - E.g.: Burst write at given rate, then process & flush at a different rate.



Latency Buffer & its access

- **Latency Buffer (LB):** A data structure implementation that temporarily stores the raw data, and has certain attributes that ensures search-ability based on a lookup criteria. A notable example for this, is the lookup based on a key, e.g.: timestamp
- **Accessor entities:**
  - Data Receiver (entity)
    - Producer: moves data to LB (thread)
    - Responsible for monitoring
    - Initiates cleanup of old data
  - Request Handler (entity)
    - Potentially overlapping data requests (threads)
    - Removes/cleans up old data (thread)
    - Other special requests (threads)



Readout requirements for buffering

Far Detector Front-end (FE) types and their characteristics:

- WIB/WIB2: Fixed arrival rate of fix sized byte arrays (payloads)
- PDs: Variable arrival rate of fix sized byte arrays (payloads)
- TPs: Variable arrival rate of variable sized byte arrays (payloads)

Other constraints:

- Arriving payloads are pre-sorted on *timestamp* field

Provide a solution for:

- Far Detector FE use cases, other use cases:
- Other use cases (e.g.: ND):
  Variable arrival rate of variable sized payloads, but **packets are not pre-sorted**!

We are looking for data structures...

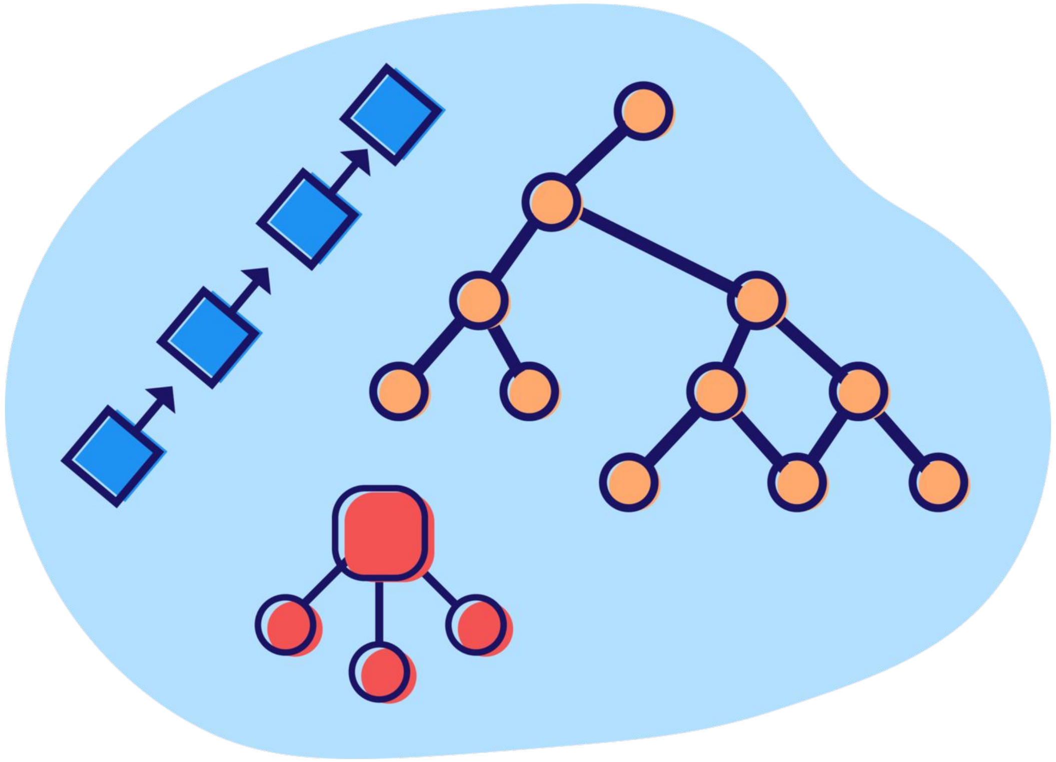
… to buffer:

- TPC data that are continuous in time and memory space
- PDS data that are non-continuous in time, but it is in memory space
- TPs that are continuous in time, but not in memory space
- ND data that are non-continuous in time, neither in memory space, non time-ordered arrival

True for all:

  - Homogeneous (won’t mix with other FE types)
  - Highly concurrent access (especially true for high-rate FEs -> TPC)

Data structures



From: https://medium.com/

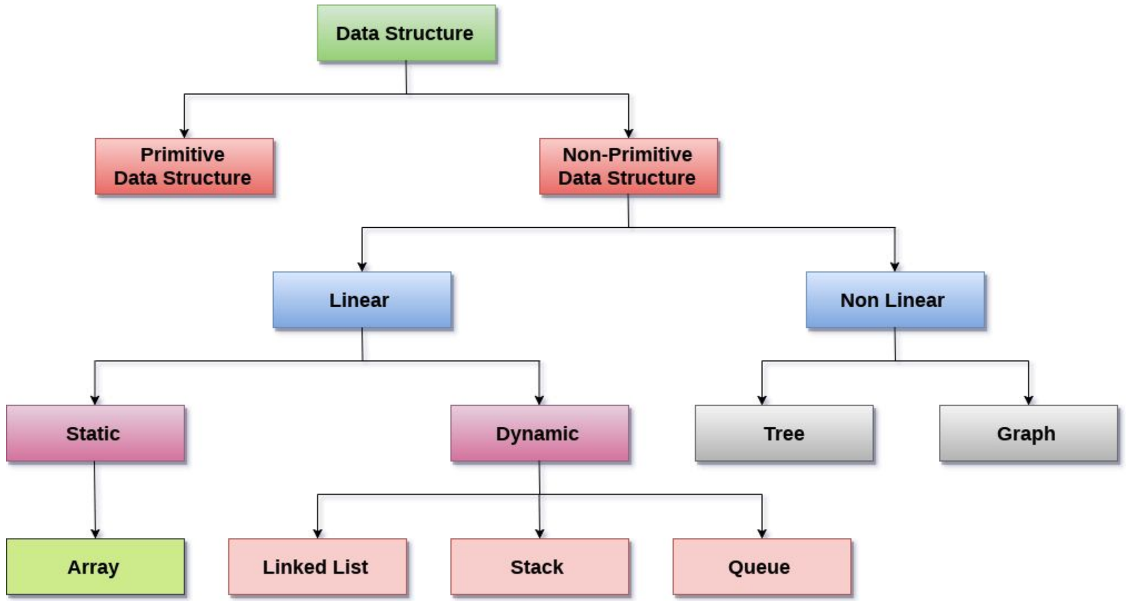
Data structures

- In computer science, a **data structure** is a data organization, management, and storage format that enabled efficient access and modification.
    - More precisely: a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.
- **Why we need them?**
    - Index- and searchability: If the processing needs to find elements efficiently
    - Concurrency: Multiple operations may happen in parallel
- **Advantages**:
    - Efficiency: Choose right data structures for the right use-cases
    - Reusability: Mature implementations exists in several different libraries
    - Abstraction: Specified by ADT, but clients use fixed interfaces

Data structure classification

Linear: All of the elements are arranged in linear order. Elements are stored in non-hierarchical way where each element has a successor and predecessor. (Except first and last.)

Non-linear: Does not form a sequence. Each element is connected with two or more other items in a non-linear arrangement. (Elements are not arranged in sequential structure.)



From: https://bcastudyguide.wordpress.com/unit-1-introduction-to-data-structure-and-its-characteristics/

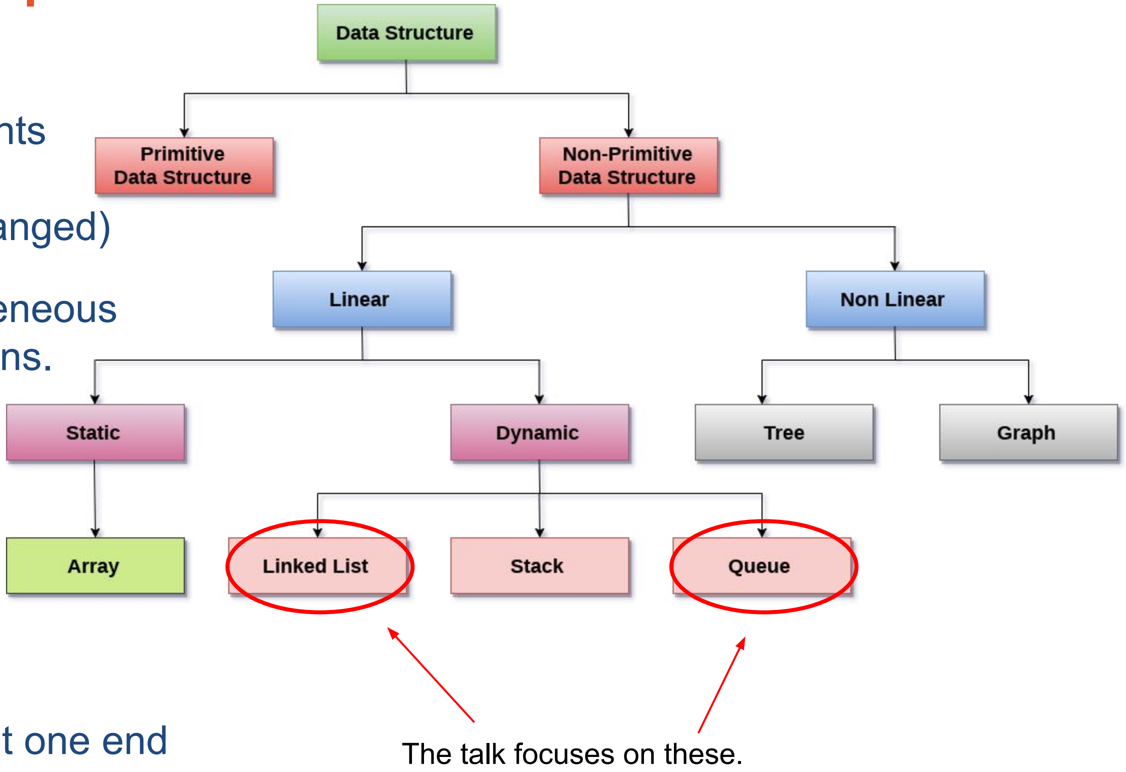
Data structure examples

Array: Collection of homogeneous elements that are contiguously arranged (mutable but static: the size cannot be changed)

List: Collection of homogeneous/heterogeneous elements stored at non-contiguous locations. Each node of the list points to its adjacent node.

Queue: Linear list in which elements can be inserted only at the "end" and deleted only at the "front". (FIFO)

Stack: Similar to queue, but only "open" at one end (*top* of the stack). (LIFO)



Concurrent access

The term **non-blocking** denotes concurrent data structures, which do not use traditional synchronization primitives like guards to ensure thread-safety.

The 3 main types of non-blocking data structures:

- **Wait-free**, if every concurrent operation is guaranteed to be finished in a finite number of steps
- **Lock-free**, if some concurrent operations are guaranteed to be finished in a finite number of steps
- **Obstruction-free**, if a concurrent operation is guaranteed to be finished in a finite number of steps, unless another concurrent operation interferes

Some data structures can only be implemented in a lock-free manner, if they are used under certain restrictions.

From Boost.Lockfree: https://www.boost.org/doc/libs/master/doc/html/lockfree.html

Implementations

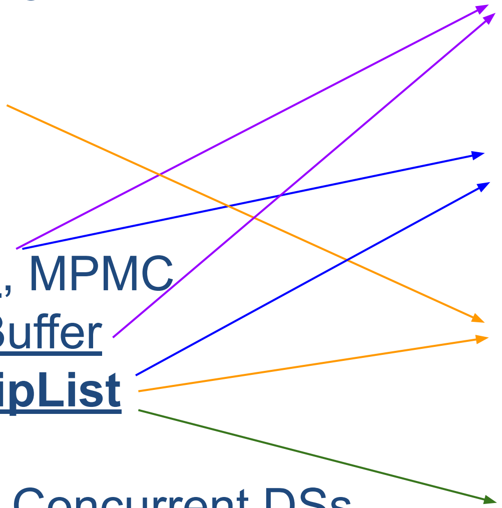
- Custom/homebrew
- Boost
  - Lockfree: ::queue, ::stack, ::spsc_queue
- TBB & oneAPI
  - Concurrent: ::hash_map, ::lru_cache, ::map, ::queue, ::vector, etc…
  - Internal _concurrent_skip_list
- Folly
  - Queues: **SPSC, MPMC (with optional blocking)** ← (The appfwk uses these for inter module communication.)
  - LockFreeRingBuffer
  - ConcurrentSkipList

(Details on the unified C++ memory model can be found under the cppreference.)

Considerations for LBs

Data structure implementations:

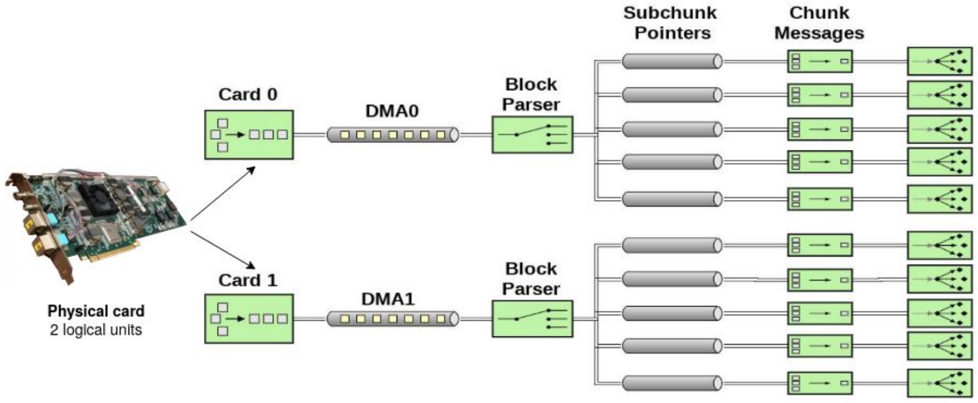
- Custom/homebrew
- Boost.Lockfree
- Folly:
  Queues: **<u>SPSC</u>**, MPMC
  <u>LockFreeRingBuffer</u>
  <u>ConcurrentSkipList</u>
- TBB & oneAPI: Concurrent DSs
  Internal _concurrent_skip_list

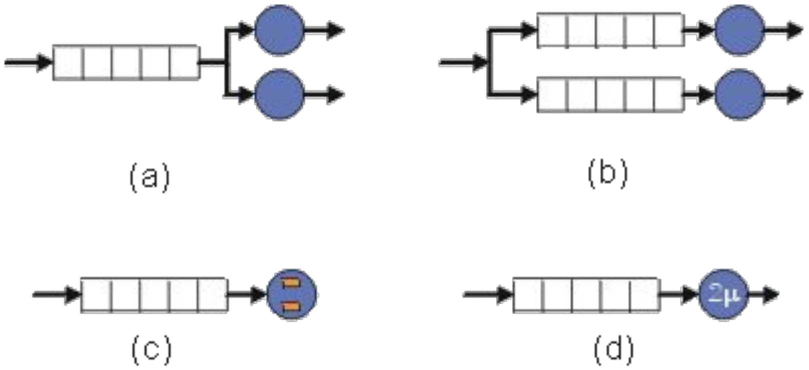


Data to buffer:

- TPC data: continuous in time and memory space. (high rate)
- PDS data: non-continuous in time, but it is in memory space. (moderate rate)
- TPs: continuous in time, but not in memory space. (low rate)
- ND data: non-continuous in time, neither in memory space, non time-ordered arrival. (low rate)

Data structures for LBs

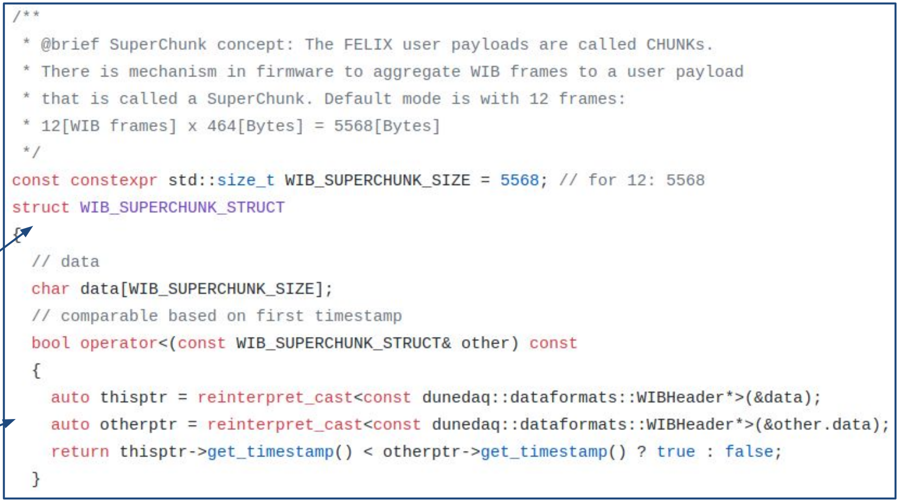




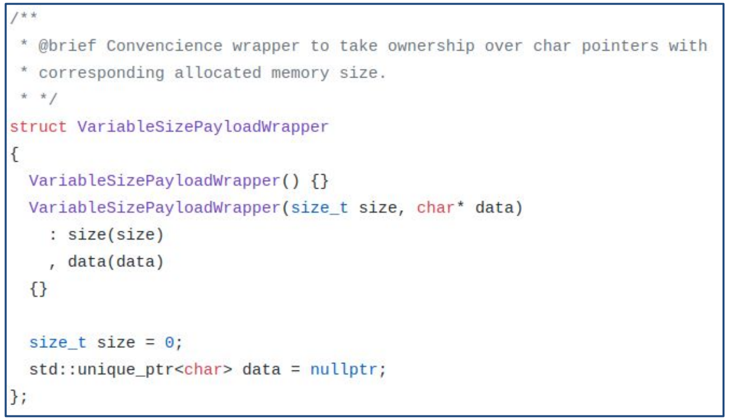
Payload structures

<u>FE payloads can be contiguously arranged.</u>

- Fix size: can be stored continuously
  - Fix rate: key can translate to position
  - Var rate: needs “find”/”search”
  - FE payloads are sortable
- Var size: elements are stored non-continuously
  - Fix rate: needs “insert”
  - Var rate: needs “insert” and “find”/”search”
  - FE types can implement own sortable wrappers with overloaded operator



```
/**
 * @brief SuperChunk concept: The FELIX user payloads are called CHUNKs.
 * There is mechanism in firmware to aggregate WIB frames to a user payload
 * that is called a SuperChunk. Default mode is with 12 frames:
 * 12[WIB frames] x 464[Bytes] = 5568[Bytes]
 */
const constexpr std::size_t WIB_SUPERCHUNK_SIZE = 5568; // for 12: 5568
struct WIB_SUPERCHUNK_STRUCT
{
  // data
  char data[WIB_SUPERCHUNK_SIZE];
  // comparable based on first timestamp
  bool operator<(const WIB_SUPERCHUNK_STRUCT& other) const
  {
    auto thisptr = reinterpret_cast<const dunedaq::dataformats::WIBHeader*>(&data);
    auto otherptr = reinterpret_cast<const dunedaq::dataformats::WIBHeader*>(&other.data);
    return thisptr->get_timestamp() < otherptr->get_timestamp() ? true : false;
  }
```

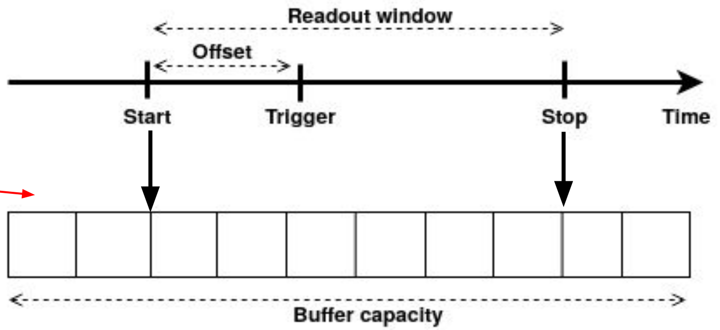


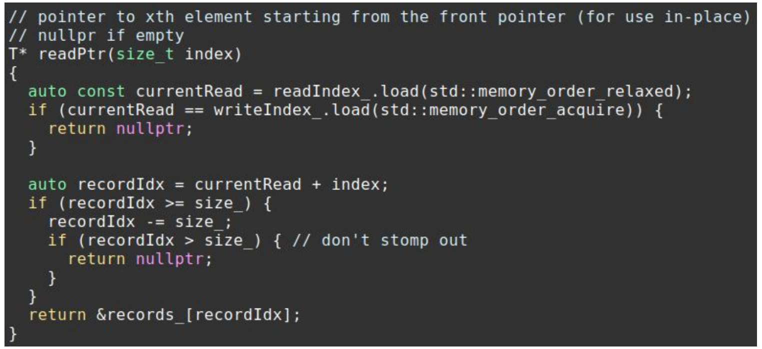
```
/**
 * @brief Convencience wrapper to take ownership over char pointers with
 * corresponding allocated memory size.
 * */
struct VariableSizePayloadWrapper
{
  VariableSizePayloadWrapper() {}
  VariableSizePayloadWrapper(size_t size, char* data)
    : size(size)
    , data(data)
  {}

  size_t size = 0;
  std::unique_ptr<char> data = nullptr;
};
```

TPC data buffering with SPSC queue

- Used in production at ProtoDUNE-SP
- Trigger timestamp can be translated to element position/index in the queue
- Modifications for DUNE in readout:
    - *AccessableSPSC* (*readPtr* function)
    - At certain occupancy threshold, a "data-cleanup" is requested
    - *RequestHandler* executes requests:
        - ClassA - data request
        - ClassB - cleanup request
    - A safe margin in form of extra payloads is needed to buffer an extra few seconds



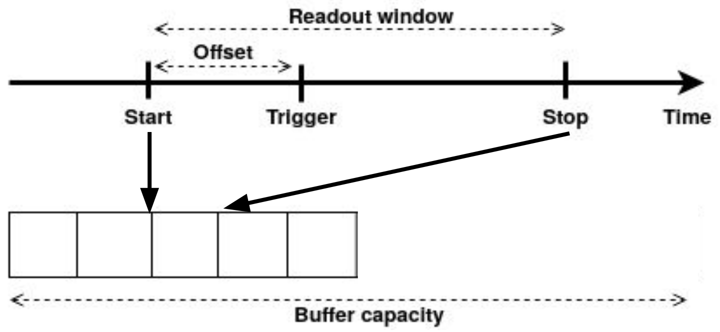


```
// pointer to xth element starting from the front pointer (for use in-place)
// nullpr if empty
T* readPtr(size_t index)
{
  auto const currentRead = readIndex_.load(std::memory_order_relaxed);
  if (currentRead == writeIndex_.load(std::memory_order_acquire)) {
    return nullptr;
  }

  auto recordIdx = currentRead + index;
  if (recordIdx >= size_) {
    recordIdx -= size_;
    if (recordIdx > size_) { // don't stomp out
      return nullptr;
    }
  }
  return &records_[recordIdx];
}
```

PDS data buffering

- **Problem: time-gaps in contiguously arranged elements:**



- **Since v2.6 there are two implementations** for the PDS LB and their corresponding request handling solutions:
  - **SPSC based**, with LB extension to ensure searchability in the buffer
  - **SkipList based**, with “find”/”search” functionality

Courtesy of Florian Till Groetschla

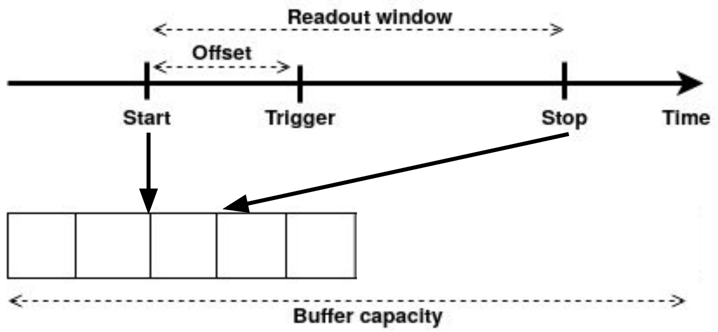
PDS data buffering with SPSC queue

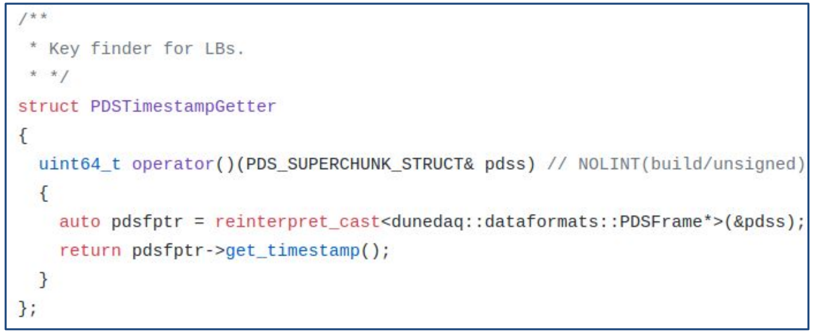
- Introduce “Key” and “KeyGetter” template Parameters to the SPSC as an extension: *SearchableProducerConsumerQueue*

- Implement a binary search algorithm in the function of the new queue with signature:

  T* find_element(Key& key) { … }

- FE types need to implement corresponding *KeyGetter* function objects.





```
/**
 * Key finder for LBs.
 * */
struct PDSTimestampGetter
{
  uint64_t operator()(PDS_SUPERCHUNK_STRUCT& pdss) // NOLINT(build/unsigned)
  {
    auto pdsfptr = reinterpret_cast<dunedaq::dataformats::PDSFrame*>(&pdss);
    return pdsfptr->get_timestamp();
  }
};
```

- RequestHandler uses Trigger TS as key for the *find_element* function.

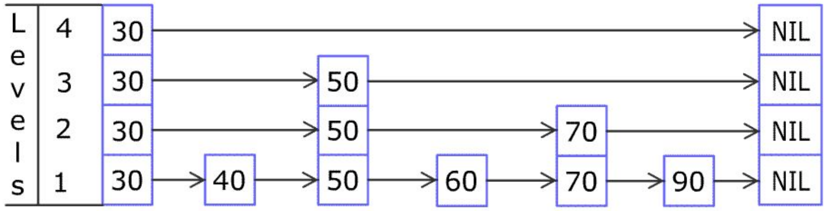
SkipList

The **SkipList is a probabilistic, ordered data structure providing O(log(n)) lookup, insertion and deletion complexity at average.**

It has the best features of a sorted array (for searching), while maintaining a linked-list structure that allows insertion (that the array does not have).

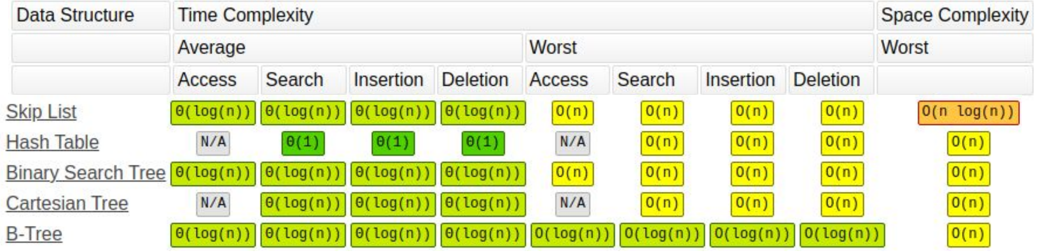
Fast search is achieved by maintaining a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one.

The subsequences support binary searching by starting at the highest subsequence and working towards the bottom, using the links to check when one should move forward in the list or down to a lower subsequence.



From Wikipedia: https://en.wikipedia.org/wiki/Skip_list

SkipList & BTree



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

- It provides it without the need for tree balancing, page splitting, etc., that are required for BTrees.
- ~ 20 years younger invention than the BTree. (1990 vs 1970s)
- BTrees made a lot of sense for databases when the data lived most of its life on disk and was moved to memory and cached for running queries on them
  - BTrees do lot of extra work to reduce disk I/O, which is needless overhead if our data fits in memory. (Which is true for the LBs.)

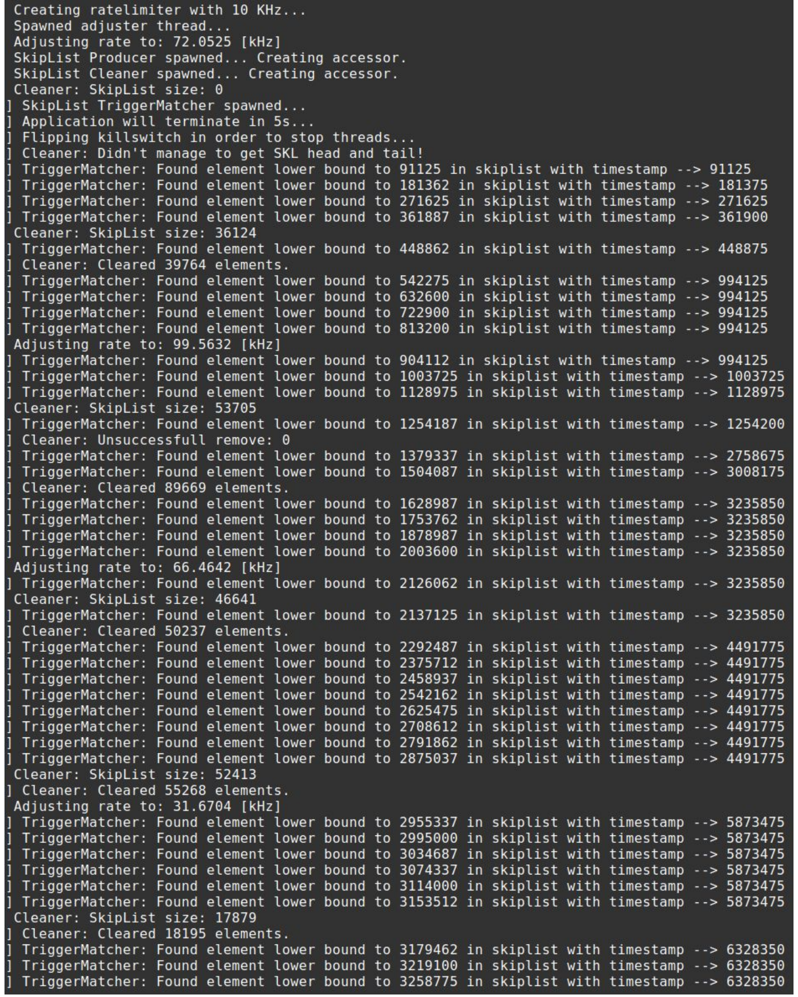
ConcurrentSkipList from Folly

The folly::ConcurrentSkipList implements A Provably Correct Scalable Concurrent Skip List.

- Good high contention performance
- Small memory overhead. (~40% less, compared with std::set)
- Read accesses are lock-free and mostly wait-free. Write access require locks, but these are local to the predecessor and/or successor nodes.
- Lazy remove with GC support. Removed nodes are deleted when the last "accessor" is destroyed.
- Can be 10x slower than std::set when the list is small (<1000 elements)
- Write operations are ~30% slower in single-threaded env compared to std::set

CSKL high contention test

- Variable rate (10-100 kHz) **producer** of 5568 Bytes payloads with timestamp increments (we always “push back”).
- **Searcher** thread looks for an element with given timestamp at 10 Hz.
- **Cleaner** thread periodically (every second) looks at tail and head timestamps. If the timestamp distance exceeds a duration, elements that are outside a time duration, are removed.



```
 Creating ratelimiter with 10 KHz...
 Spawned adjuster thread...
 Adjusting rate to: 72.0525 [kHz]
 SkipList Producer spawned... Creating accessor.
 SkipList Cleaner spawned... Creating accessor.
 Cleaner: SkipList size: 0
] SkipList TriggerMatcher spawned...
] Application will terminate in 5s...
] Flipping killswitch in order to stop threads...
] Cleaner: Didn't manage to get SKL head and tail!
] TriggerMatcher: Found element lower bound to 91125 in skiplist with timestamp --> 91125
] TriggerMatcher: Found element lower bound to 181362 in skiplist with timestamp --> 181375
] TriggerMatcher: Found element lower bound to 271625 in skiplist with timestamp --> 271625
] TriggerMatcher: Found element lower bound to 361887 in skiplist with timestamp --> 361900
 Cleaner: SkipList size: 36124
] TriggerMatcher: Found element lower bound to 448862 in skiplist with timestamp --> 448875
] Cleaner: Cleared 39764 elements.
] TriggerMatcher: Found element lower bound to 542275 in skiplist with timestamp --> 994125
] TriggerMatcher: Found element lower bound to 632600 in skiplist with timestamp --> 994125
] TriggerMatcher: Found element lower bound to 722900 in skiplist with timestamp --> 994125
] TriggerMatcher: Found element lower bound to 813200 in skiplist with timestamp --> 994125
 Adjusting rate to: 99.5632 [kHz]
] TriggerMatcher: Found element lower bound to 904112 in skiplist with timestamp --> 994125
] TriggerMatcher: Found element lower bound to 1003725 in skiplist with timestamp --> 1003725
] TriggerMatcher: Found element lower bound to 1128975 in skiplist with timestamp --> 1128975
 Cleaner: SkipList size: 53705
] TriggerMatcher: Found element lower bound to 1254187 in skiplist with timestamp --> 1254200
] Cleaner: Unsuccessfull remove: 0
] TriggerMatcher: Found element lower bound to 1379337 in skiplist with timestamp --> 2758675
] TriggerMatcher: Found element lower bound to 1504087 in skiplist with timestamp --> 3008175
] Cleaner: Cleared 89669 elements.
] TriggerMatcher: Found element lower bound to 1628987 in skiplist with timestamp --> 3235850
] TriggerMatcher: Found element lower bound to 1753762 in skiplist with timestamp --> 3235850
] TriggerMatcher: Found element lower bound to 1878987 in skiplist with timestamp --> 3235850
] TriggerMatcher: Found element lower bound to 2003600 in skiplist with timestamp --> 3235850
 Adjusting rate to: 66.4642 [kHz]
] TriggerMatcher: Found element lower bound to 2126062 in skiplist with timestamp --> 3235850
 Cleaner: SkipList size: 46641
] TriggerMatcher: Found element lower bound to 2137125 in skiplist with timestamp --> 3235850
] Cleaner: Cleared 50237 elements.
] TriggerMatcher: Found element lower bound to 2292487 in skiplist with timestamp --> 4491775
] TriggerMatcher: Found element lower bound to 2375712 in skiplist with timestamp --> 4491775
] TriggerMatcher: Found element lower bound to 2458937 in skiplist with timestamp --> 4491775
] TriggerMatcher: Found element lower bound to 2542162 in skiplist with timestamp --> 4491775
] TriggerMatcher: Found element lower bound to 2625475 in skiplist with timestamp --> 4491775
] TriggerMatcher: Found element lower bound to 2708612 in skiplist with timestamp --> 4491775
] TriggerMatcher: Found element lower bound to 2791862 in skiplist with timestamp --> 4491775
] TriggerMatcher: Found element lower bound to 2875037 in skiplist with timestamp --> 4491775
 Cleaner: SkipList size: 52413
] Cleaner: Cleared 55268 elements.
 Adjusting rate to: 31.6704 [kHz]
] TriggerMatcher: Found element lower bound to 2955337 in skiplist with timestamp --> 5873475
] TriggerMatcher: Found element lower bound to 2995000 in skiplist with timestamp --> 5873475
] TriggerMatcher: Found element lower bound to 3034687 in skiplist with timestamp --> 5873475
] TriggerMatcher: Found element lower bound to 3074337 in skiplist with timestamp --> 5873475
] TriggerMatcher: Found element lower bound to 3114000 in skiplist with timestamp --> 5873475
] TriggerMatcher: Found element lower bound to 3153512 in skiplist with timestamp --> 5873475
 Cleaner: SkipList size: 17879
] Cleaner: Cleared 18195 elements.
] TriggerMatcher: Found element lower bound to 3179462 in skiplist with timestamp --> 6328350
] TriggerMatcher: Found element lower bound to 3219100 in skiplist with timestamp --> 6328350
] TriggerMatcher: Found element lower bound to 3258775 in skiplist with timestamp --> 6328350
```

PDS data buffering with CSKL

- Introduce “less” operator for the FE payload structures.

- Add *SkipListLatencyBufferModel* with:
    - “Insert” that moves new elements into the list.
    - “Read” that returns *lower_bound* element (i.e.: closest timestamp)

- Implement *PDSRequestHandler* that need special access to the SkipList.
    - Override data request function with using Accessors to iterate and collect elements that are included in the trigger window.
    - Override cleanup request function that removes elements that are out of bound from a time-duration (e.g.: last 2 seconds based on newest timestamp)

Other potential use-cases

- Near Detector FE types
  - Variable rate and variable size -> linked list fits well
  - No guarantee of time-ordered payload arrival -> needs sorting on insertion

- Event Building
  - Sorting on insert operations
  - Composite keys make the SkipList a powerhouse

Outlook

- Investigate RingBuffer (potentially eliminates the need for “cleanup”)
- Investigate “smart” data structures. I.e.: add functionality on top of original implementations (Like the SearchableSPSC.)
- Performance evaluation of different implementations used for different use-cases

End

Thank you for your attention!